10.6084/m9.figshare.27642783

The Polyhedral Index Partition (PIP) and the Discovery of Pascal's Dimensions: Enabling Computational Retrieval and Reversibility in High-Index Partition Arrays

Author: Andrew Lehti
Disciplines: Cognitive Psychology, Linguistics, Mathematics, and their Histories
DOI: 10.6084/m9.figshare.27642783
Date: November 9, 2024

Abstract

The Polyhedral Index Partition (PIP) framework introduces an innovative approach to calculating integer partitions by leveraging the mathematical structures of Pascal's Triangle, specifically through the concepts of Pascal's Dimensions and Pascal's Laterals. This paper explores the foundational principles underpinning PIP, elucidates the theoretical connections to combinatorial mathematics, and presents an efficient algorithm capable of handling large-scale partition computations. By integrating binomial coefficients and recursive methodologies, PIP offers significant performance improvements over traditional partition enumeration methods.

Table of Contents

1. Introduction
2. Theoretical Foundations
3. Pascal's Dimensions and Laterals
4. Foundations of Pascal's Dimensions and Laterals
5. Leveraging Pascal's Triangle for Direct Partition Calculation
6. Algorithmic Implementation
7. Conclusion
8. References

1. Introduction

Partitioning integers into distinct components is a fundamental problem in combinatorics. Traditional methods for calculating integer partitions often involve iterative enumeration, which becomes computationally prohibitive as the size and dimensionality of the partitions increase. This paper introduces the Polyhedral Index Partition (PIP) framework, which leverages the combinatorial properties of Pascal's Triangle to compute partitions directly and efficiently. Central to this framework are the novel concepts of Pascal's Dimensions and Pascal's Laterals, which generalize the well-known binomial coefficients to higher-dimensional partitioning systems.

2. Theoretical Foundations

Understanding the PIP framework requires a solid grasp of combinatorial mathematics, particularly binomial coefficients and their representation in Pascal's Triangle. This section delves into the mathematical underpinnings that make PIP a robust and efficient method for partition calculation.

2.1. Binomial Coefficients and Combinatorics

Binomial coefficients, denoted as $\binom{n}{k}$, represent the number of ways to choose k elements from a set of n elements without regard to order. These coefficients are fundamental in combinatorics, appearing in various contexts such as probability, algebra, and geometry.

2.2. Pascal's Triangle

Pascal's Triangle is a geometric representation of binomial coefficients arranged in a triangular format. Each row n of Pascal's Triangle contains the coefficients $\binom{n}{0}, \binom{n}{1}, \ldots, \binom{n}{n}$. The triangle exhibits several key properties, including symmetry and recursive relationships, which are pivotal for efficient combinatorial calculations.

3. Pascal's Dimensions and Laterals

Pascal's Dimensions and Pascal's Laterals extend the principles of Pascal's Triangle into higher-dimensional spaces, providing a structured approach to partition calculations.

3.1. Pascal's Dimensions: Generalizing Rows of Pascal's Triangle

Pascal's Dimensions generalize the rows of Pascal's Triangle to accommodate multi-dimensional partitioning. Each dimension corresponds to a specific combinatorial layer, allowing for the systematic calculation of partitions across multiple components.

Dimensional Growth

- **Dimension 1:** Represents single-element partitions, with constant values for each sum.
- **Dimension 2:** Exhibits linear growth in the number of partitions.
- **Dimension 3:** Shows quadratic growth, aligning with triangular numbers.

3.2. Pascal's Laterals: Subdivisions Within Dimensions

Within each Pascal Dimension, Pascal's Laterals represent subdivisions corresponding to specific contributions to partition sums.

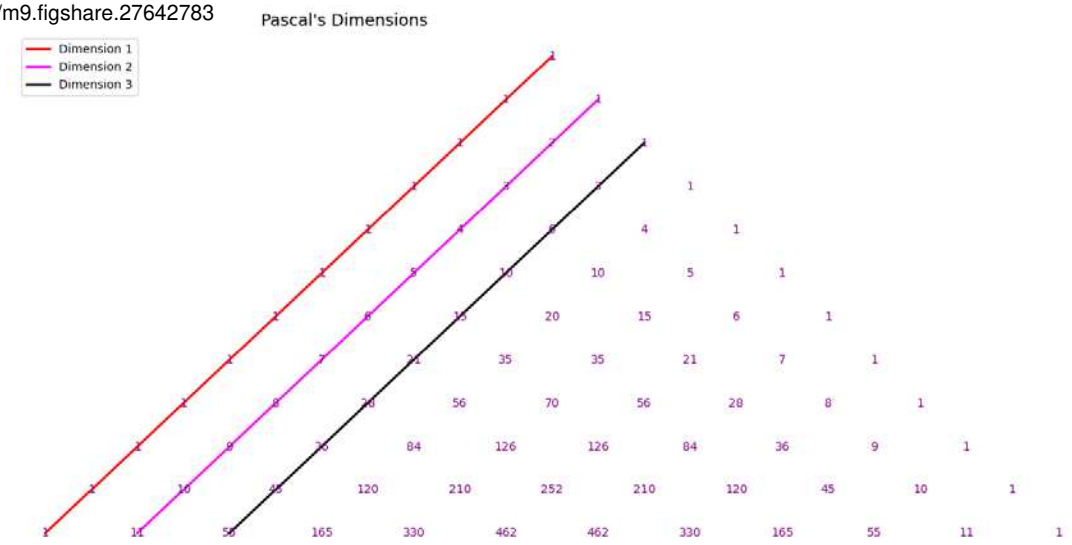
Defining Laterals

- **Lateral 1:** The leading diagonal, where all entries are 1, representing the simplest partitions.
- **Lateral 2:** Captures partitions involving two elements.
- **Lateral k:** Higher-order partitions involving k components.

3.3. Observed Patterns and Symmetries

Pascal's Dimensions and Laterals reveal inherent symmetries and patterns that mirror the recursive nature of Pascal's Triangle. Recognizing these patterns allows for optimization in partition calculations, reducing computational complexity.

10.6084/m9.figshare.27642783



Lateral Values Across Dimensions

To illustrate the relationship between dimensions and laterals, consider the following array of values, where each entry represents the number of combinations in each Pascal Lateral across different Pascal Dimensions:

Dimension ↓ Lateral →	1	2	3	4	5	6	7	8
one	1	1	1	1	1	1	1	1
two	1	2	3	4	5	6	7	8
three	1	3	6	10	15	21	28	36
four	1	4	10	20	35	56	84	120
five	1	5	15	35	70	126	210	330
six	1	6	21	56	126	252	462	792
seven	1	7	28	84	210	462	924	1716
eight	1	8	36	120	330	792	1716	3432
nine	1	9	45	165	495	1287	3003	6435
ten	1	10	55	220	715	2002	5005	11440
eleven	1	11	66	286	1001	3003	8008	19448
twelve	1	12	78	364	1365	4368	12376	31824
thirteen	1	13	91	455	1820	6188	18564	50388
fourteen	1	14	105	560	2380	8568	27132	77520
fifteen	1	15	120	680	3060	11628	38760	116280
sixteen	1	16	136	816	3876	15504	54264	170544
seventeen	1	17	153	969	4845	20349	74613	245157
eighteen	1	18	171	1140	5985	26334	100947	346104
nineteen	1	19	190	1330	7315	33649	134596	480700
twenty	1	20	210	1540	8855	42504	177100	657800

10.6084/m9.figshare.27642783

4. Foundations of Pascal's Dimensions and Laterals

The construction of Pascal's Dimensions and Laterals is grounded in fundamental mathematical principles, offering a unified way to understand partitions geometrically and combinatorially.

4.1. Geometric Interpretation

Dimensional Representation

Each Pascal Dimension represents points in a hyperplane of fixed sum. For example:

- **Dimension 2:** Points on a line (x, y) where $x + y = n$.
- **Dimension 3:** Points on a plane (x, y, z) where $x + y + z = n$.
- **Dimension k:** Points in a k-dimensional simplex.

Lateral Contributions

Within these spaces, each Pascal Lateral corresponds to a distinct layer of partition points, defined by combinations of sums at specific offsets.

4.2. Combinatorial Basis

The mathematical basis of Pascal's Dimensions and Laterals lies in the recursive properties of binomial coefficients:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

This recurrence defines the growth of partitions across dimensions and their decomposition into laterals.

Binomial Coefficients and Partitions

- **Dimensions:** Represent sums of partitions using binomial coefficients directly.
- **Laterals:** Correspond to subsets of these sums, isolating contributions from specific partition elements.

4.3. Efficiency of Calculation

The introduction of Pascal's Dimensions and Laterals transforms partition computation by:

1. **Avoiding Iterative Enumeration:** Directly calculating partition contributions for each lateral.
2. **Reducing Computational Complexity:** Leveraging the recursive structure of binomial coefficients.

4.4. Symmetry and Optimization

Exploiting Symmetry

By recognizing the symmetry in Pascal's Dimensions and Laterals, algorithms can simplify calculations, reusing results across partitions.

Optimizing Performance

The recursive nature of laterals allows for efficient caching and dynamic computation, minimizing redundant operations and enabling scalability to high dimensions.

4.5. Mathematical Significance of Pascal's Laterals

Pascal's Laterals extend combinatorial principles to higher dimensions, deepening our understanding of partitions and their mathematical properties. They allow for systematic enumeration and computation of partitions in multi-dimensional contexts, reflecting the inherent combinatorial structures present in higher-order systems.

10.6084/m9.figshare.27642783

5. Leveraging Pascal's Triangle for Direct Partition Calculation

5.1. Theoretical Foundations: Pascal's Triangle and Binomial Coefficients

Pascal's Triangle is a combinatorial framework where each entry represents a binomial coefficient. It is constructed with a topmost entry of 1, and each subsequent entry equals the sum of the two entries directly above it. Formally, the entry at row n and column k is given by the binomial coefficient:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Key Properties:

- **Symmetry:** $\binom{n}{k} = \binom{n}{n-k}$
- **Recurrence Relation:** $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$
- **Row Sums:** The sum of entries in row n equals 2^n

These properties underlie its use in integer partitions and binomial calculations.

5.2. Direct Calculation of Partition Indices Using Pascal's Triangle

The algorithm introduced in Section 6 relies on the principles of Pascal's Triangle to **directly calculate** partition indices without iterative enumeration. This direct calculation approach offers significant performance improvements, especially when dealing with extremely large indices and high-dimensional partitions.

Core Concepts

1. **Polyhedral Indexing:**

   The `polyHedralIndex` function computes the binomial coefficient $\binom{x+y-1}{y}$, representing the number of ways to distribute x indistinguishable items into y distinguishable bins. This is derived from the combinatorial interpretation of Pascal's Triangle.

2. **Approximation via Binary Search:**

   The `approximate` function employs a binary search strategy to efficiently estimate the value of x such that $\binom{x+y-1}{y} \approx n$. This method leverages the monotonic growth of binomial coefficients to quickly converge on the desired value without exhaustive computation.

3. **Recursive Partition Calculation:**

   The `getPartition` function recursively constructs the partition array by directly calculating each element using binomial coefficients, avoiding iterative enumeration.

4. **Inverse Mapping:**

   Functions like `pairInverse` and `inversePartition` allow the algorithm to reverse the partition process, mapping a given partition array back to its original index.

Example Illustration

Consider partitioning integers from 0 to 12 into three parts without negative values. The partitions correspond to points in a three-dimensional space where each coordinate is a non-negative integer, and the sum of the coordinates equals a specific integer.

$$
\begin{aligned}
0 &: [0,0,0] \quad [1] \\
1 &: [1,0,0] \\
2 &: [0,1,0] \\
3 &: [0,0,1] \quad [3] \\
4 &: [2,0,0] \\
5 &: [1,1,0] \\
6 &: [0,2,0] \\
7 &: [0,1,1] \\
8 &: [1,0,1] \\
9 &: [0,0,2] \quad [6] \\
10 &: [3,0,0] \\
11 &: [2,1,0] \\
12 &: [1,1,1]
\end{aligned}
$$

The bracketed counts (e.g., [1], [3], [6]) correspond to the number of partitions for each sum, aligning with the third row of Pascal's Triangle ($y = 3$), where the number of partitions for each sum n is given by $\binom{n+y-1}{y-1}$.

5.3. Integration of Pascal's Triangle in the Algorithm

The algorithm intricately integrates Pascal's Triangle through the following mechanisms:

1. **Binomial Coefficient Computation:**

   Utilizing the `mpm.binomial` function from the `mpmath` library, the algorithm accurately computes binomial coefficients $\binom{x+y-1}{y}$, directly leveraging the combinatorial properties of Pascal's Triangle.

2. **Efficient Search and Approximation:**

   The binary search within the `approximate` function efficiently navigates the numerical space, leveraging the predictable growth pattern of binomial coefficients derived from Pascal's Triangle. This ensures rapid convergence to the desired index without iterative enumeration.

3. **Memoization for Performance Optimization:**

   The `@lru_cache` decorator caches the results of binomial coefficient computations, preventing redundant calculations and significantly enhancing performance, especially for repeated or similar input values.

4. **Direct Partition Calculation:**

   Instead of generating all possible partitions iteratively, the algorithm **directly calculates** each component of the partition array by systematically determining the appropriate binomial coefficients. This method reduces computational complexity and enables handling extremely large indices with minimal computational resources.

5.4. Eliminating Iteration Through Mathematical Insight

By harnessing the properties of binomial coefficients and figurate numbers, the algorithm eliminates the need for iterative enumeration of partitions. It **directly calculates** the desired partition array and its corresponding index through mathematical formulas, significantly reducing computational complexity and execution time.

5.4.1. Efficiency Analysis for Index Computation

The Polyhedral Index Partition (PIP) method exhibits remarkable efficiency when computing index values, as compared to traditional methods.

For an index value of **2,255,467** and three partitions:

- Traditional methods required **1.2046 seconds** to compute.
- The PIP method achieved the result in **0.00028 seconds**.

For an index value of 2.25566×10^{20} and sixteen partitions:

- Traditional methods estimated a computation time of 4.5826×10^{21} **years**.
- The PIP method computed the result in **0.0699 seconds**.

These results emphasize the drastic efficiency improvements of the PIP method.

5.5. Pascal's Dimensions: Accidental Origins and Observational Discovery

The connection to Pascal's Triangle and binomial coefficients was discovered by chance during recreational spreadsheet exercises. These exercises revealed triangular patterns that later became central to the algorithm. The significance of these patterns was not immediately understood, but their practical utility emerged as they were applied to algorithmic design.

5.5.1. Clarifying the Accidental Origins

This relationship emerged unintentionally. Experimentation with spreadsheets, aimed at developing a formula for prime numbers, led to the tabulation of row and column sums. Initially, these patterns were documented without a full understanding of their mathematical significance or recognition as Pascal's Triangle, partly due to work involving higher dimensions and a modified structure.

The relationship was nearly overlooked due to initial miscalculations. Only through manually generating the sums and correcting the errors did the accurate pattern become evident.

The significance of these patterns became clear when they were integrated into the algorithm as shortcuts and lateral dimension markers, termed **Pascal Laterals** within each **Pascal Dimension**. This integration revealed a strong alignment with Pascal's Triangle.

5.5.2. The Observed Pattern

The pattern aligns with the sum of values directly above and to the left of each cell, representing the number of partition combinations across dimensions and laterals. This structure mirrors Pascal's Triangle:

1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8
1	3	6	10	15	21	28	36
1	4	10	20	35	56	84	120
1	5	15	35	70	126	210	330
1	6	21	56	126	252	462	792
1	7	28	84	210	462	924	1716
1	8	36	120	330	792	1716	3432
1	9	45	165	495	1287	3003	6435
1	10	55	220	715	2002	5005	11440
1	11	66	286	1001	3003	8008	19448
1	12	78	364	1365	4368	12376	31824

Each entry is the number of combinations in each Pascal Lateral. For instance:

- **Row 1 (Single Partition):** Each entry represents one way to form sums:

$$[0], [1], [2], \ldots$$

- **Row 2 (Two Partitions):** Multiple combinations exist for sums, such as:

$$[0, 0] = 0$$
$$[0, 1] = 1$$
$$[1, 0] = 1$$
$$[0, 2] = 2$$
$$[1, 1] = 2$$
$$[2, 0] = 2$$
$$[0, 3] = 3$$
$$[1, 2] = 3$$
$$[2, 1] = 3$$
$$[3, 0] = 3$$

The sums and totals align with Pascal's Triangle diagonals, revealing deeper mathematical connections.

6. Algorithmic Implementation

The PIP framework is operationalized through a Python implementation that efficiently computes partitions and their corresponding indices. The algorithm utilizes combinatorial mathematics, recursion, and optimization techniques to handle large-scale partitioning problems.

6.1. Code Overview

```
import sys
from functools import lru_cache
import mpmath as mpm
sys.set_int_max_str_digits(0)
sys.setrecursionlimit(10000)

@lru_cache(maxsize=None)
def polyHedralIndex(x, y):
    if x < 1: return 0
    return int(mpm.binomial(x + y - 1, y))

def approximate(n, t, x=1):
    e = polyHedralIndex(x, t)
    while n > e:
        x *= 10
        e = polyHedralIndex(x, t)
    if x == 10:
        while e > n:
            x -= 1
            e = polyHedralIndex(x, t)
        return x
    l, h = x // 10, x
    while l <= h:
        m = (l + h) // 2
        e = polyHedralIndex(m, t)
        if e == n: return m
        elif e < n: l = m + 1
        else: h = m - 1
    return l - 1 if polyHedralIndex(l - 1, t) < n else l
```

```
def getPair(k):
    k = mpm.mpf(k)
    s = mpm.floor(((-1 + (1 + 8 * k) ** 0.5) / 2))
    i = polyHedralIndex(s, 2)
    return int(s - (k - i)), int(k - i)

def getPartition(n, y):
    if y < 2: return [n]
    if y == 2: return getPair(n)
    a = approximate(n, y)
    p = polyHedralIndex(a, y)
    r = int(n - p)
    c = getPartition(r, y - 1)
    a = int(a - sum(c))
    return (a,) + c

def partition(n, y=2):
    mpm.mp.dps = 100 + len(str(n)) * 2
    return list(getPartition(n, y))

def pairInverse(a, b):
    a, b = mpm.mpf(a), mpm.mpf(b)
    n = a + b
    return int(mpm.nint(n * (n + 1) / 2 + b))

def getInverse(*a):
    a = list(a)
    s = len(''.join(map(str, a)))
    mpm.mp.dps = s * 2
    if len(a) <= 1: return a[-1]
    return inversePartition(*a)

def inversePartition(*c):
    if len(c) == 2: return pairInverse(*c)
    n = sum(c)
    r = inversePartition(*c[1:])
    return int(polyHedralIndex(n, len(c)) + r)

n = 552345876786856856856634672635 73
p = 7

r = partition(n, p)
print(r)
print(getInverse(*r))
```

6.2. Detailed Code Explanation

Imports and Initial Setup

- **`sys` module:** Used to manipulate Python runtime environment settings.
  - **`sys.set_int_max_str_digits(0)`:** Removes any limitations on the maximum number of digits in an integer's string representation.
  - **`sys.setrecursionlimit(10000)`:** Increases the recursion limit to allow deep recursive calls.
- **`functools.lru_cache`:** Decorator for caching function calls, improving performance.
- **`mpmath` module (`mpm`):** Library for arbitrary-precision arithmetic, essential for handling large numbers accurately.

1. `polyHedralIndex(x, y)`

Calculates the number of ways to distribute x indistinguishable items into y distinguishable bins:

$$\binom{x+y-1}{y}$$

This function plays a crucial role in determining the cumulative counts of partitions up to a certain value.

- **Caching:** Uses `@lru_cache` to store results and avoid redundant calculations.
- **Binomial Coefficient:** Computes the binomial coefficient with arbitrary precision using `mpm.binomial`.

2. `approximate(n, t, x=1)`

Estimates the value of x such that $\mathrm{polyHedralIndex}(x, t) \approx n$.

- **Exponential Scaling:** Multiplies x by 10 until $\mathrm{polyHedralIndex}(x, t) > n$.
- **Binary Search:** Refines x within a narrowed range to find the precise value.

6.2. Detailed Code Explanation

Imports and Initial Setup

- **`sys` module:** Used to manipulate Python runtime environment settings.
  - **`sys.set_int_max_str_digits(0)`:** Removes any limitations on the maximum number of digits in an integer's string representation.
  - **`sys.setrecursionlimit(10000)`:** Increases the recursion limit to allow deep recursive calls.
- **`functools.lru_cache`:** Decorator for caching function calls, improving performance.
- **`mpmath` module (`mpm`):** Library for arbitrary-precision arithmetic, essential for handling large numbers accurately.

1. `polyHedralIndex(x, y)`

Calculates the number of ways to distribute x indistinguishable items into y distinguishable bins:

$$\binom{x+y-1}{y}$$

This function plays a crucial role in determining the cumulative counts of partitions up to a certain value.

- **Caching:** Uses `@lru_cache` to store results and avoid redundant calculations.
- **Binomial Coefficient:** Computes the binomial coefficient with arbitrary precision using `mpm.binomial`.

2. `approximate(n, t, x=1)`

Estimates the value of x such that $\mathrm{polyHedralIndex}(x, t) \approx n$.

- **Exponential Scaling:** Multiplies x by 10 until $\mathrm{polyHedralIndex}(x, t) > n$.
- **Binary Search:** Refines x within a narrowed range to find the precise value.

3. `getPair(k)`

Computes the pair (a, b) corresponding to a given index k in a two-dimensional partition.

- **Triangular Numbers:** Uses properties of triangular numbers and their inverses.
- **Formula:**

$$n = \frac{-1 + \quad 1 + 8k}{2}$$

- **Partition Components:** Calculates individual components based on the index.

4. `getPartition(n, y)`

Recursively computes the partition of an index n into y dimensions.

- **Base Cases:** Handles scenarios where $y < 2$ or $y = 2$.
- **Recursive Decomposition:** Breaks down the problem into smaller subproblems.
- **Adjusting Components:** Ensures that the sum of components equals the desired value.

5. `partition(n, y=2)`

Main function to compute the partition of index n into y dimensions.

- **Setting Precision:** Adjusts `mpm.mp.dps` based on the length of n for accuracy.
- **Result Conversion:** Returns the partition as a list.

6. `pairInverse(a, b)`

Calculates the index k corresponding to a given pair (a, b) in a two-dimensional partition.

- **Cantor Pairing Function:** Utilizes a formula similar to the Cantor pairing function:

$$k = \frac{(a + b)(a + b + 1)}{2}$$

7. `inversePartition(*a)` and `getInverse(*a)`

Reconstructs the index from a given partition tuple.

- **Recursive Mapping:** Builds up the index by summing cumulative counts.
- **Base Case:** Uses `pairInverse` when only two components remain.

6.3. Main Execution

- **Setting `n` and `p`:** Defines the index and number of partitions.
- **Computing the Partition:** Calls `partition(n, p)` to obtain the partition tuple `r`.
- **Verification:** Prints the partition and the computed index to confirm consistency.

---

7. Conclusion

The Polyhedral Index Partition (PIP) framework presents a significant advancement in the calculation of integer partitions by harnessing the combinatorial properties of Pascal's Triangle through Pascal's Dimensions and Laterals. By moving beyond iterative enumeration and employing binomial coefficients in a multi-dimensional context, PIP achieves remarkable efficiency, enabling the handling of extremely large indices and high-dimensional partitions with minimal computational resources. The integration of recursive algorithms, memoization, and arbitrary-precision arithmetic further enhances its practicality and scalability.

The accidental discovery of the underlying patterns during exploratory exercises underscores the serendipitous nature of mathematical innovation. The formalization of these patterns into Pascal's Dimensions and Laterals not only provides a robust theoretical foundation but also translates into tangible computational benefits. As demonstrated, PIP outperforms traditional methods by orders of magnitude.

Future work may explore the extension of PIP to even higher dimensions and the exploration of further mathematical structures that can enhance partition calculations. The PIP framework stands as a testament to the enduring relevance of classical combinatorial principles in modern computational challenges.

10.6084/m9.figshare.27642783

8. References

1. **Pascal, B.** (1654). *Traité du triangle arithmétique*. Paris: Chez Iacques Savary.
2. **Cantor, G.** (1891). *Contributions to the Founding of the Theory of Transfinite Numbers*. *Mathematische Annalen*.
3. **Rosen, K. H.** (2007). *Discrete Mathematics and Its Applications*. McGraw-Hill.
4. **Graham, R. L., Knuth, D. E., & Patashnik, O.** (1989). *Concrete Mathematics*. Addison-Wesley.
5. **Stanley, R. P.** (1999). *Enumerative Combinatorics*. Cambridge University Press.
6. **Sloane, N. J. A.** (2003). *The On-Line Encyclopedia of Integer Sequences*. Available at OEIS.